Managing I/O: CFRunLoop and CFStream

Session 808

Managing I/O: CFRunLoop and CFStream

Becky Willrich
Application Frameworks

Managing I/O: The Problem

- I/O is slow and lengthy
  - Speed is bounded by the pipe
  - Requires relatively little CPU
- Want to use the CPU for other processing
  - Avoid starving one pipe waiting on another
  - Allow speedy user event processing
- One solution is multi-threading . . .
- But threading is complex

Managing I/O: The Dream

- Let “someone else” deal with the pipe
- You get informed when processing is needed
- You handle I/O in small bursts amidst other events
    - You know nothing else is running
    - No contention for your in-memory objects!

Managing I/O: The Solution

- Have a multiplexer that shares CPU between multiple inputs
- Each input triggered only if there is work to be done
- When an input is triggered, it alone gets processing time

The Multiplexer: CFRunLoop

- CFRunLoop is the multiplexer
- The inputs are CFRunLoopSources
- Pre-defined sources for common inputs
- New sources can be programmatically created

Managing I/O

- CFRunLoop
  - What is it?
  - Common sources: timers, message ports, sockets
- CFStream
  - What is it?
  - How does it progress through its lifetime?
  - How do I use it with the run loop?

CFRunLoop

Doug Davidson
Application Frameworks

CFRunLoop

- Enables advanced services
- Generalized event manager and dispatcher
- Maintains sets of event sources
- Callbacks to registered handlers when events occur
    - While the run loop is running

As the Loop Runs

- Run loop waits
- Something happens
- Source is triggered
- Client receives callback
- Callback returns, run loop waits again

As the Loop Runs

- Run loop waits
- Something happens
- Source is triggered
- Client receives callback
- Callback returns, run loop waits again

As the Loop Runs

- Run loop waits
- Something happens
- Source is triggered
- Client receives callback
- Callback returns, run loop waits again

As the Loop Runs

- Run loop waits
- Something happens
- Source is triggered
- Client receives callback
- Callback returns, run loop waits again

As the Loop Runs

- Run loop waits
- Something happens
- Source is triggered
- Client receives callback
- Callback returns, run loop waits again

Run Loops

- Each thread has exactly one run loop
    - Created automatically for you
- CFRunLoop is CoreFoundation interface to it
- Apps will usually use Carbon Events or AppKit interfaces
- Use the highest-level API that meets your needs
- CFRunLoop for CoreFoundation-based sources

Who Runs the Run Loop?

- Carbon: RunApplicationEventLoop()
- Cocoa: NSApplication
- CoreFoundation, or secondary threads:
    - CFRunLoopRun()
    - CFRunLoopRunInMode()

Run Loop Modes

- Run loop event-source sets are called “modes”
- Modes are used to restrict which event sources are monitored

  Example: **NSModalPanelRunLoopMode**
- Normally you will use and run the run loop in the default mode
- Can also use “common” modes

  kCFRunLoopCommonModes

CFRunLoop Sources

- Some provided source types
  - CFRunLoopTimer
  - CFMessagePort
  - CFMachPort
  - CFSocket
  - CFStream
- You can create your own, but it can be complex

CFRunLoop Sources

- Allow you to attach a context, an arbitrary pointer of your choice
    - With callbacks for retain, release, etc.
- Should be invalidated when no longer in use
    - Release alone usually is not enough
- Ordered for priority when there is contention

CFRunLoopTimers

- One-shot or repeating
- Callback function provided at create time
- Add it to a run loop with modes
- When date arrives, it will fire
    - But only when the run loop is running
- If one-shot, automatically invalidated
- Fire date can be changed on the fly

Using CFRunLoopTimer

- Create timer

```
CFRunLoopTimerRef timer = CFRunLoopTimerCreate(NULL,
   CFAbsoluteTimeGetCurrent(), 1.0, 0, 0, myCallBack, NULL);
```

- Add to run loop

```
CFRunLoopRef loop = CFRunLoopGetCurrent();
CFRunLoopAddTimer(loop, timer,
   kCFRunLoopCommonModes);
```

- Run run loop (if no-one is running it for you)

```
CFRunLoopRun();
```

Using CFRunLoopTimer

- Create timer

```
CFRunLoopTimerRef timer = CFRunLoopTimerCreate(NULL,
   CFAbsoluteTimeGetCurrent(), 1.0, 0, 0, myCallBack, NULL);
```

- Add to run loop

```
CFRunLoopRef loop = CFRunLoopGetCurrent();
CFRunLoopAddTimer(loop, timer,
   kCFRunLoopCommonModes);
```

- Run run loop (if no-one is running it for you)

```
CFRunLoopRun();
```

Using CFRunLoopTimer

- Create timer

```
CFRunLoopTimerRef timer = CFRunLoopTimerCreate(NULL,
   CFAbsoluteTimeGetCurrent(), 1.0, 0, 0, myCallBack, NULL);
```

- Add to run loop

```
CFRunLoopRef loop = CFRunLoopGetCurrent();
CFRunLoopAddTimer(loop, timer,
   kCFRunLoopCommonModes);
```

- Run run loop (if no-one is running it for you)

```
CFRunLoopRun();
```

CFMessagePort

- Low-overhead, high-performance IPC
- Lower-level than Apple Events
- Between threads or processes on the same machine
- Messages can be sent
    - One-way and asynchronous
    - With reply and synchronous

CFMessagePort

- Local port is your end
  `CFMessagePortCreateLocal()`
- Local ports may be named and advertised with a CFString
- Remote port is the other end, looked up by name
  `CFMessagePortCreateRemote()`

Sending Messages

- Messages are a CFData and integer message ID
- Optional replies are CFData
- Server defines and publishes the protocol for messages
    - The format of the data
    - What the message IDs mean
    - If there will be a reply

Messaging Process

- Client sends a message
  `CFMessagePortSendRequest()`
- Server's callback is called
- Server returns optional reply
- Client gets reply
  `CFMessagePortSendRequest()` returns
- Can optionally receive other messages or events while waiting

CFMachPort

- Allows Mach port to serve as run loop source
- Access to raw Mach messages
- Lower-level than CFMessagePort
- Handles receive end only

CFMachPort

- Create with existing Mach port
  `CFMachPortCreateWithPort()`
- Or let it create the Mach port for you
  `CFMachPortCreate()`
- You get a callback when a message arrives

CFSocket

- Allows BSD socket to serve as run loop source
- Not a complete socket wrapper
- Notifications for read, write, accept, connect
- Handles arbitrary flavors of sockets
    - TCP, UDP, IPv4, IPv6, local, etc.

CFSocket

- Create with existing socket
  `CFSocketCreateWithNative()`
- Or let it create the socket for you
  `CFSocketCreate()`
  `CFSocketCreateWithSocketSignature()`
  `CFSocketCreateConnectedToSocketSignature()`
- Choose what callbacks you want:
  - Read, data, accept, connect, write

Using CFSocket

- Create socket

```
struct sockaddr_in a = {0, AF_INET, 1234, 0};
CFDataRef d = CFDataCreate(NULL, (UInt8 *)&a,
   sizeof(struct sockaddr_in));
CFSocketSignature signature =
  {PF_INET, SOCK_STREAM, IPPROTO_TCP, d};
CFSocketRef socket =
   CFSocketCreateWithSocketSignature(NULL,
   &signature, kCFSocketAcceptCallBack,
   acceptConnection, NULL);
```

Using CFSocket

- Create socket

```
struct sockaddr_in a = {0, AF_INET, 1234, 0};
CFDataRef d = CFDataCreate(NULL, (UInt8 *)&a,
   sizeof(struct sockaddr_in));
CFSocketSignature signature =
  {PF_INET, SOCK_STREAM, IPPROTO_TCP, d};
CFSocketRef socket =
   CFSocketCreateWithSocketSignature(NULL,
   &signature, kCFSocketAcceptCallBack,
   acceptConnection, NULL);
```

Using CFSocket

- Create socket

```
struct sockaddr_in a = {0, AF_INET, 1234, 0};
CFDataRef d = CFDataCreate(NULL, (UInt8 *)&a,
   sizeof(struct sockaddr_in));
CFSocketSignature signature =
  {PF_INET, SOCK_STREAM, IPPROTO_TCP, d};
CFSocketRef socket =
   CFSocketCreateWithSocketSignature(NULL,
   &signature, kCFSocketAcceptCallBack,
   acceptConnection, NULL);
```

Using CFSocket

- Create socket

```
struct sockaddr_in a = {0, AF_INET, 1234, 0};
CFDataRef d = CFDataCreate(NULL, (UInt8 *)&a,
   sizeof(struct sockaddr_in));
CFSocketSignature signature =
  {PF_INET, SOCK_STREAM, IPPROTO_TCP, d};
CFSocketRef socket =
   CFSocketCreateWithSocketSignature(NULL,
   &signature, kCFSocketAcceptCallBack,
   acceptConnection, NULL);
```

Using CFSocket

- Create source, add to run loop, run

```
CFRunLoopSourceRef source =
  CFSocketCreateRunLoopSource(NULL,
  socket, 0);
CFRunLoopRef loop = CFRunLoopGetCurrent();
CFRunLoopAddSource(loop, source,
  kCFRunLoopDefaultMode);
CFRunLoopRun();
```

Using CFSocket

- Create source, add to run loop, run

```
CFRunLoopSourceRef source =
  CFSocketCreateRunLoopSource(NULL,
  socket, 0);
CFRunLoopRef loop = CFRunLoopGetCurrent();
CFRunLoopAddSource(loop, source,
  kCFRunLoopDefaultMode);
CFRunLoopRun();
```

Using CFSocket

- Create source, add to run loop, run

```
CFRunLoopSourceRef source =
  CFSocketCreateRunLoopSource(NULL,
  socket, 0);
CFRunLoopRef loop = CFRunLoopGetCurrent();
CFRunLoopAddSource(loop, source,
  kCFRunLoopDefaultMode);
CFRunLoopRun();
```

Using CFSocket

```
acceptConnection() {
        CFSocketRef child =
          CFSocketCreateWithNative(NULL,
          *(CFSocketNativeHandle *)data,
          kCFSocketDataCallBack, receiveData, NULL);
        CFRunLoopSourceRef childSource =
          CFSocketCreateRunLoopSource(NULL, child, 0);
        CFRunLoopRef loop = CFRunLoopGetCurrent();
        CFRunLoopAddSource(loop, childSource,
          kCFRunLoopDefaultMode);
        CFRelease(childSource);
        }
```

Using CFSocket

```
acceptConnection() {
        CFSocketRef child =
          CFSocketCreateWithNative(NULL,
          *(CFSocketNativeHandle *)data,
          kCFSocketDataCallBack, receiveData, NULL);
        CFRunLoopSourceRef childSource =
          CFSocketCreateRunLoopSource(NULL, child, 0);
        CFRunLoopRef loop = CFRunLoopGetCurrent();
        CFRunLoopAddSource(loop, childSource,
          kCFRunLoopDefaultMode);
        CFRelease(childSource);
        }
```

Using CFSocket

```
acceptConnection() {
        CFSocketRef child =
          CFSocketCreateWithNative(NULL,
          *(CFSocketNativeHandle *)data,
          kCFSocketDataCallBack, receiveData, NULL);
        CFRunLoopSourceRef childSource =
          CFSocketCreateRunLoopSource(NULL, child, 0);
        CFRunLoopRef loop = CFRunLoopGetCurrent();
        CFRunLoopAddSource(loop, childSource,
          kCFRunLoopDefaultMode);
        CFRelease(childSource);
        }
```

Using CFSocket

```
receiveData() {
        static char helloWorld[] = “HTTP/1.0 200
           OK\r\n\r\nhello, world\r\n”;
        CFDataRef response = CFDataCreate(NULL,
           helloWorld, strlen(helloWorld));
        CFSocketSendData(child, NULL, response, 0.0);
        CFRelease(response);
        CFSocketInvalidate(child);
        CFRelease(child);
        }
```

Using CFSocket

```
receiveData() {
        static char helloWorld[] = “HTTP/1.0 200
           OK\r\n\r\nhello, world\r\n”;
        CFDataRef response = CFDataCreate(NULL,
           helloWorld, strlen(helloWorld));
        CFSocketSendData(child, NULL, response, 0.0);
        CFRelease(response);
        CFSocketInvalidate(child);
        CFRelease(child);
        }
```

Using CFSocket

```
receiveData() {
        static char helloWorld[] = “HTTP/1.0 200
          OK\r\n\r\nhello, world\r\n”;
        CFDataRef response = CFDataCreate(NULL,
          helloWorld, strlen(helloWorld));
        CFSocketSendData(child, NULL, response, 0.0);
        CFRelease(response);
        CFSocketInvalidate(child);
        CFRelease(child);
        }
```

Using CFSocket

```
receiveData() {
        static char helloWorld[] = “HTTP/1.0 200
           OK\r\n\r\nhello, world\r\n”;
        CFDataRef response = CFDataCreate(NULL,
           helloWorld, strlen(helloWorld));
        CFSocketSendData(child, NULL, response, 0.0);
        CFRelease(response);
        CFSocketInvalidate(child);
        CFRelease(child);
        }
```

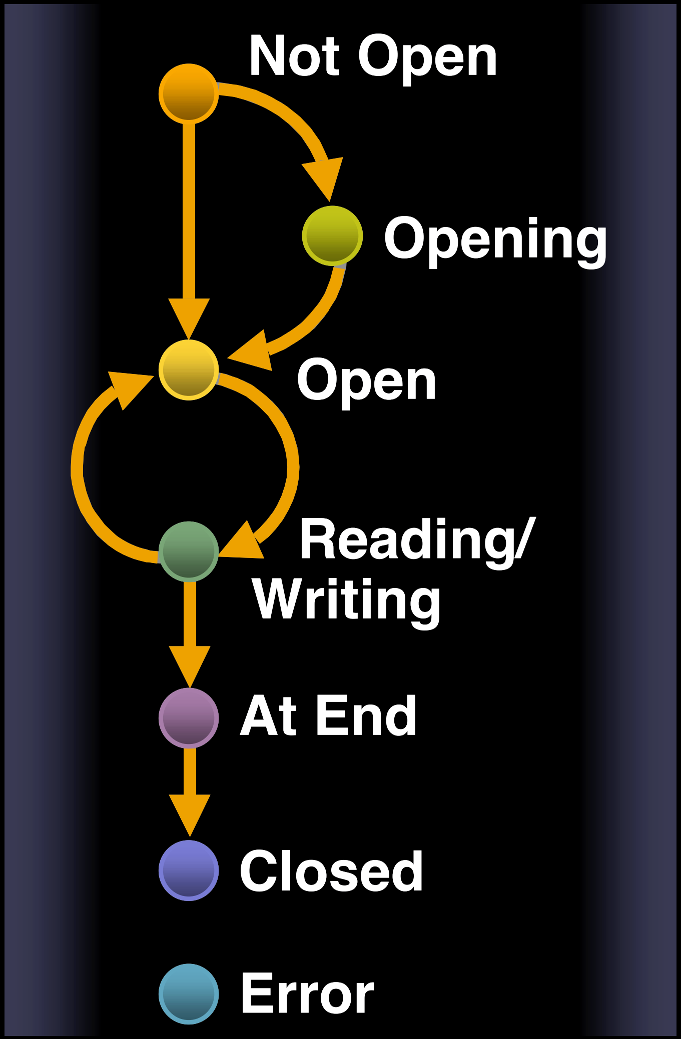
CFStream

Becky Willrich
Application Frameworks

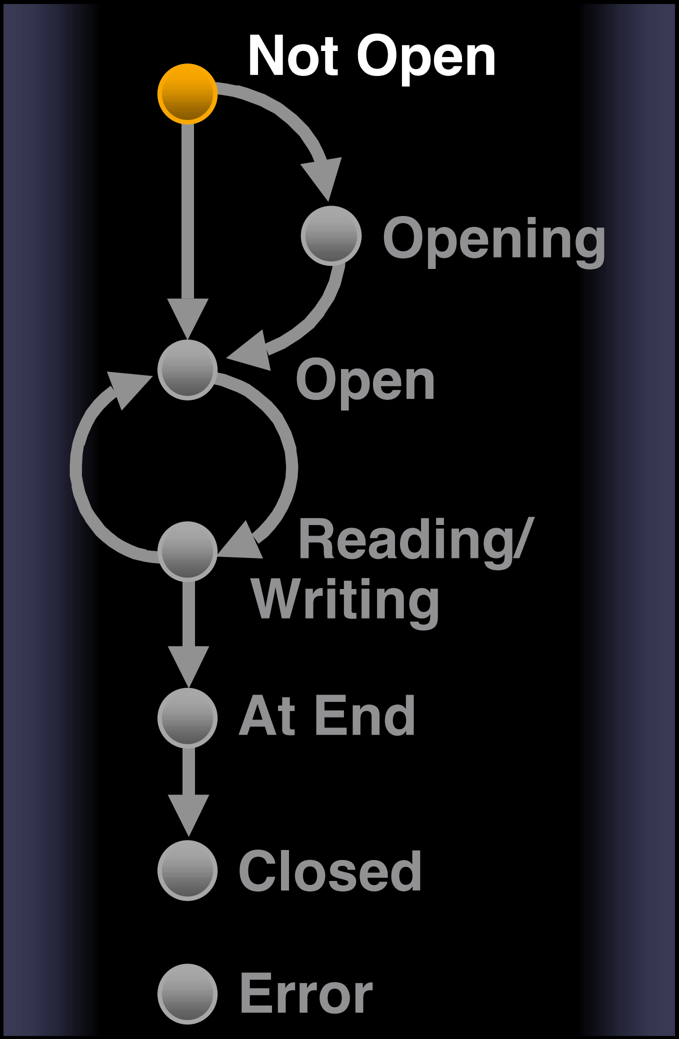
What Is a CFStream?

- One-directional stream of bytes
- Abstracts away the byte source/destination
- Two CFTypes: CFReadStream and CFWriteStream
- Moves through several states in its lifetime

A Stream’s Lifetime

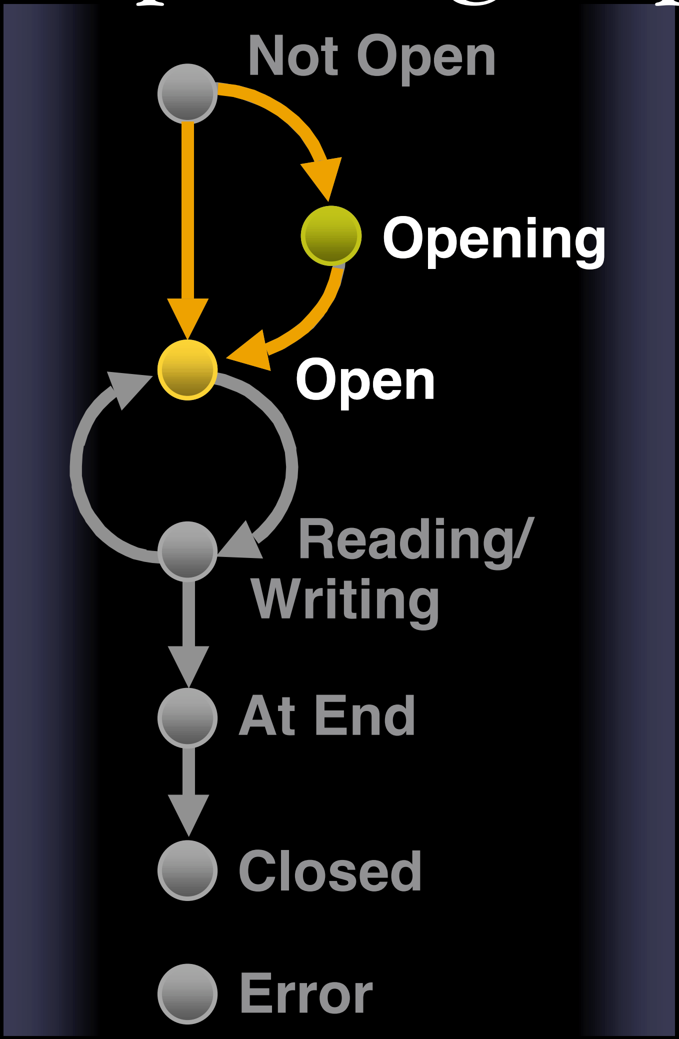


A Stream's Lifetime–Not Open



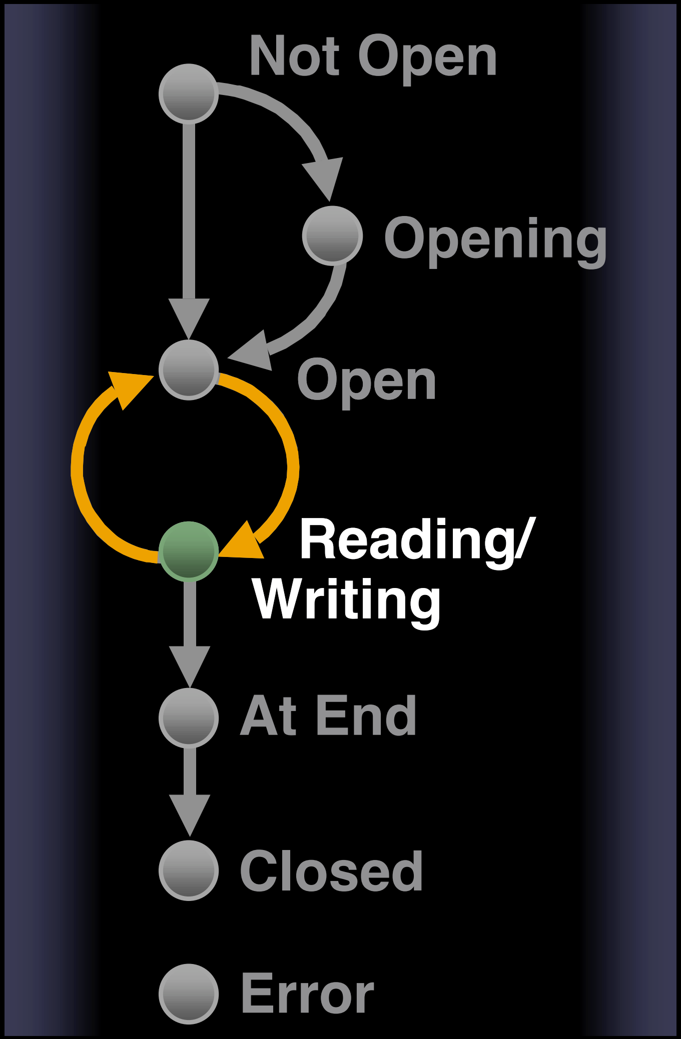
- Stream exists only in memory
- No system resources allocated yet
- Usually configure the stream at this time
- Leave this state by calling CFReadStreamOpen()

A Stream's Lifetime–Opening/Open



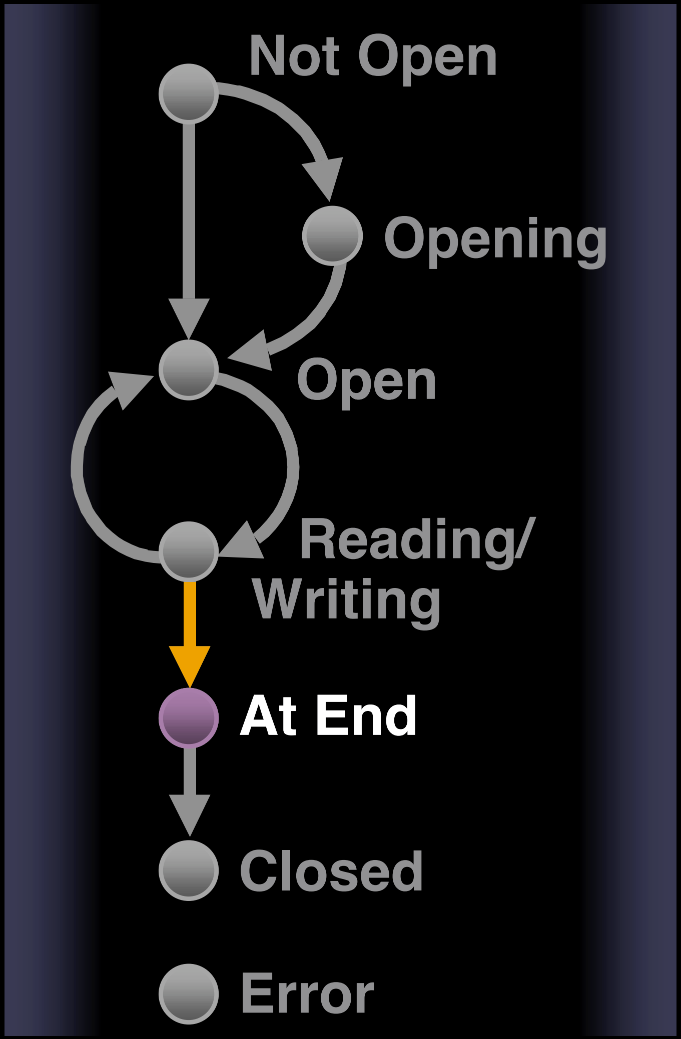
- Streams reserve their system resources when they are opening
- Open may not finish immediately
- State "Opening" used until the open is complete

A Stream’s Lifetime–Read/Write



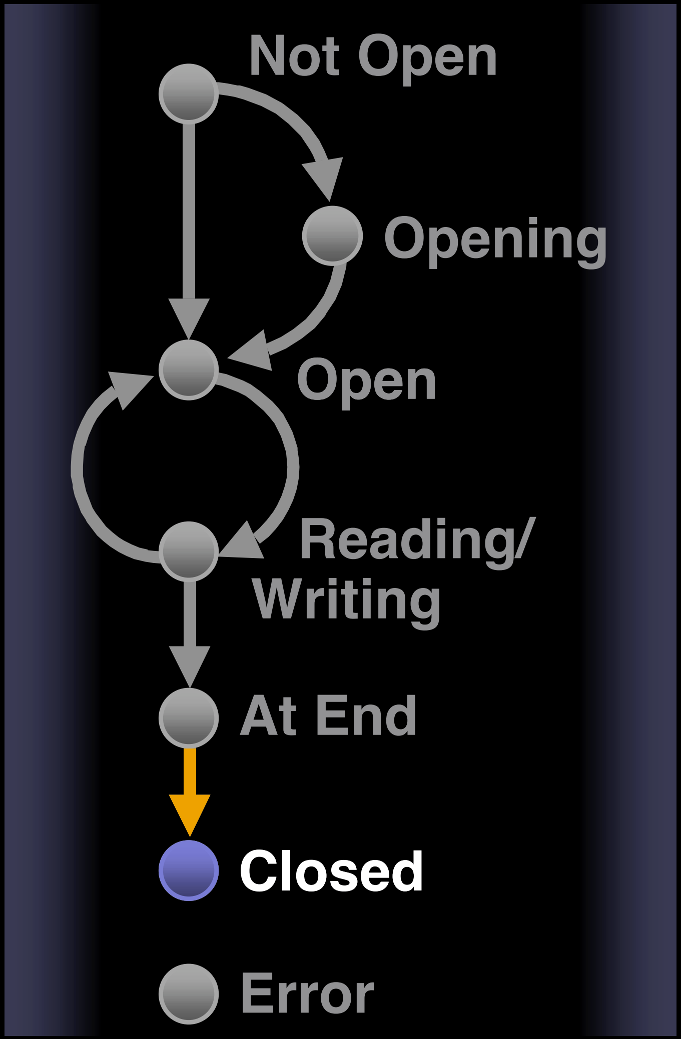
- Once a stream is open, you may call CFReadStreamRead() or CFWriteStreamWrite()
- Returns
  - Number of bytes read/written
  - 0 if at end
  - -1 if an error occurred
- Don’t have to wait for the open to complete before calling read/write
  - But you will block!

A Stream’s Lifetime—AtEnd



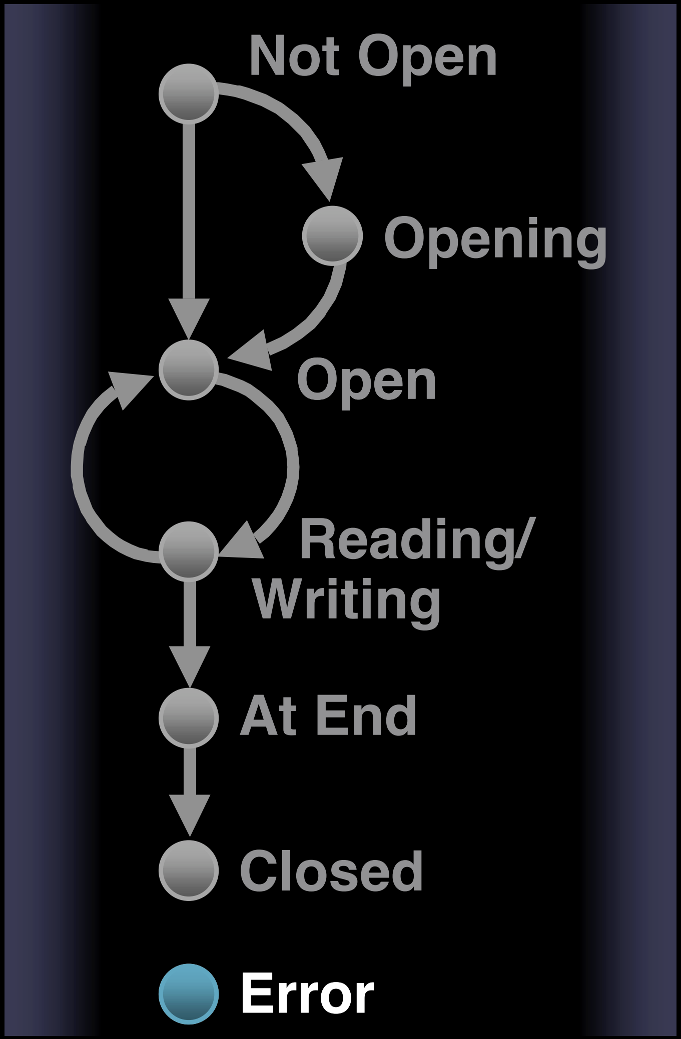
- A read stream has been completely emptied
- A write stream has been completely filled
- No more bytes will ever be read/written

A Stream's Lifetime—Closed



- Happens after you close the stream
- System resources have been dismissed
- You can still query properties of the stream

A Stream's Lifetime—Error



- Errors can happen at any point in the stream's lifetime
- All errors are fatal (the stream cannot be restored to another state)
- Use CFReadStreamGetError() to diagnose or report the error

Creating a CFStream

- Custom creation functions for each kind of stream
- CF can create streams to/from:
  - Files
  - Sockets
  - Memory
- CFNetwork adds HTTP

Using a CFStream

- Event driven
- Blocking
- Polling

Event Driven

- The recommended way to use streams
- Set a client on the stream
    - A client is a context pointer with a callback function
- Schedule the stream on a run loop
- Open the stream
- Handle events in the callback

Setting the Client

```
void configureStream(CFReadStreamRef stream) {

    CFStreamClientContext context = {0,
        CFArrayCreateMutable(kCFAllocatorDefault, 0,
            &kCFTypeArrayCallBacks),
        CFRetain, CFRelease, CFCopyDescription};

    CFOptionFlags events =
        kCFStreamEventOpenCompleted |
        kCFStreamEventHasBytesAvailable |
        kCFStreamEventEndEncountered |
        kCFStreamEventErrorOccurred;

    CFReadStreamSetClient(stream, events,
        handleEvent, &context);
}
```

Setting the Client

```
void configureStream(CFReadStreamRef stream) {

    CFStreamClientContext context = {0,
        CFArrayCreateMutable(kCFAllocatorDefault, 0,
            &kCFTypeArrayCallBacks),
        CFRetain, CFRelease, CFCopyDescription};

    CFOptionFlags events =
        kCFStreamEventOpenCompleted |
        kCFStreamEventHasBytesAvailable |
        kCFStreamEventEndEncountered |
        kCFStreamEventErrorOccurred;

    CFReadStreamSetClient(stream, events,
        handleEvent, &context);
}
```

Setting the Client

```
void configureStream(CFReadStreamRef stream) {

    CFStreamClientContext context = {0,
        CFArrayCreateMutable(kCFAllocatorDefault, 0,
            &kCFTypeArrayCallBacks),
        CFRetain, CFRelease, CFCopyDescription};

    CFOptionFlags events =
        kCFStreamEventOpenCompleted |
        kCFStreamEventHasBytesAvailable |
        kCFStreamEventEndEncountered |
        kCFStreamEventErrorOccurred;

    CFReadStreamSetClient(stream, events,
        handleEvent, &context);
}
```

Opening the Stream

```
void openStream(CFReadStreamRef stream) {

    CFReadStreamScheduleWithRunLoop(stream,
        CFRunLoopGetCurrent(),
        kCFRunLoopCommonModes);

    CFReadStreamOpen(stream);
}
```

Handling Events

```
void handleEvent(CFReadStreamRef stream,
    CFStreamEventType event, void *info) {

    CFStringRef string;
    CFMutableArrayRef array = (CFMutableArrayRef)info;

    string = CFStringCreateWithFormat(kCFAllocatorDefault,
        NULL, CFSTR(“Event %d”), event);
    CFArrayAppend(array, string);
    CFRelease(string);

    if (event == kCFStreamEventHasBytesAvailable) {
        UInt8 buf[SIZE];
        int num = CFReadStreamRead(stream, buf, SIZE);
    }
}
```

Handling Events

```
void handleEvent(CFReadStreamRef stream,
    CFStreamEventType event, void *info) {

    CFStringRef string;
    CFMutableArrayRef array = (CFMutableArrayRef)info;

    string = CFStringCreateWithFormat(kCFAllocatorDefault,
        NULL, CFSTR("Event %d"), event);
    CFArrayAppend(array, string);
    CFRelease(string);

    if (event == kCFStreamEventHasBytesAvailable) {
        UInt8 buf[SIZE];
        int num = CFReadStreamRead(stream, buf, SIZE);
    }
}
```

Handling Events

```
void handleEvent(CFReadStreamRef stream,
    CFStreamEventType event, void *info) {

    CFStringRef string;
    CFMutableArrayRef array = (CFMutableArrayRef)info;

    string = CFStringCreateWithFormat(kCFAllocatorDefault,
        NULL, CFSTR("Event %d"), event);
    CFArrayAppend(array, string);
    CFRelease(string);

    if (event == kCFStreamEventHasBytesAvailable) {
        UInt8 buf[SIZE];
        int num = CFReadStreamRead(stream, buf, SIZE);
    }
}
```

Event Flow

- OpenCompleted, then zero or more HasBytesAvailable or CanAcceptBytes, then AtEnd
- ErrorOccurred can happen at any time and will always be the last event
- Once you receive HasBytesAvailable, no more will be sent until you read!
    - Same is true for CanAcceptBytes and writing
- You can change the client at any time, but look out for race conditions

Blocking

- Open the stream
- Read or write until done
  - Each call blocks until at least one byte is read or written
  - Each call then processes as much as possible without blocking
- Dispose

Blocking—Example

```
if (!CFReadStreamOpen(stream)) {
    handleError(stream);
} else {
    int numBytes;
    UInt8 buf[BUFSIZE];
    do {
        numBytes = CFReadStreamRead(stream,
            buf, BUFSIZE);
        processBytes(buf, numBytes);
    } while (numBytes > 0);

    if (numBytes < 0) {
        handleError(stream);
    }
}
CFReadStreamClose(stream);
```

Blocking—Example

```
if (!CFReadStreamOpen(stream)) {
    handleError(stream);
} else {
    int numBytes;
    UInt8 buf[BUFSIZE];
    do {
        numBytes = CFReadStreamRead(stream,
            buf, BUFSIZE);
        processBytes(buf, numBytes);
    } while (numBytes > 0);

    if (numBytes < 0) {
        handleError(stream);
    }
}
CFReadStreamClose(stream);
```

Blocking—Example

```
if (!CFReadStreamOpen(stream)) {
    handleError(stream);
} else {
    int numBytes;
    UInt8 buf[BUFSIZE];
    do {
        numBytes = CFReadStreamRead(stream,
            buf, BUFSIZE);
        processBytes(buf, numBytes);
    } while (numBytes > 0);

    if (numBytes < 0) {
        handleError(stream);
    }
}
CFReadStreamClose(stream);
```

Blocking—Example

```
if (!CFReadStreamOpen(stream)) {
    handleError(stream);
} else {
    int numBytes;
    UInt8 buf[BUFSIZE];
    do {
        numBytes = CFReadStreamRead(stream,
            buf, BUFSIZE);
        processBytes(buf, numBytes);
    } while (numBytes > 0);

    if (numBytes < 0) {
        handleError(stream);
    }
}
CFReadStreamClose(stream);
```

Polling

- Open the stream
- When you want to poll, call
  - CFReadStreamHasBytesAvailable(stream)
  - CFWriteStreamCanAcceptBytes(stream)
- If TRUE is returned, you can read or write without blocking
- Check for errors separately
  - CFRead/WriteStreamGetStatus(stream)
  - CFRead/WriteStreamGetError(stream)

Polling—Example

```
// Assumes the stream is already open
void pollStream(CFReadStreamRef stream) {

    if (CFReadStreamHasBytesAvailable(stream)) {

        UInt8 buf[BUFSIZE];
        int numBytes = CFReadStreamRead(stream, buf, BUFSIZE);

        if (numBytes > 0)
            processBytes(buf, numBytes);
        if (numBytes == 0)
            finishUp();
        else if (numBytes < 0)
            handleError();

    } else if (CFReadStreamGetStatus(stream) ==
        kCFStreamStatusError) {
        handleError();
    }
}
```

Polling—Example

```
// Assumes the stream is already open
void pollStream(CFReadStreamRef stream) {

    if (CFReadStreamHasBytesAvailable(stream)) {

        UInt8 buf[BUFSIZE];
        int numBytes = CFReadStreamRead(stream, buf, BUFSIZE);

        if (numBytes > 0)
            processBytes(buf, numBytes);
        if (numBytes == 0)
            finishUp();
        else if (numBytes < 0)
            handleError();

    } else if (CFReadStreamGetStatus(stream) ==
        kCFStreamStatusError) {
        handleError();
    }
}
```

Polling—Example

```
// Assumes the stream is already open
void pollStream(CFReadStreamRef stream) {

    if (CFReadStreamHasBytesAvailable(stream)) {

        UInt8 buf[BUFSIZE];
        int numBytes = CFReadStreamRead(stream, buf, BUFSIZE);

        if (numBytes > 0)
            processBytes(buf, numBytes);
        if (numBytes == 0)
            finishUp();
        else if (numBytes < 0)
            handleError();

    } else if (CFReadStreamGetStatus(stream) ==
        kCFStreamStatusError) {
        handleError();
    }
}
```

Stream Properties

- Properties represent attributes of the stream not related to the actual bytes
- Setting properties configures the stream
- Getting properties gets out-of-band information about the stream
- Property names are CFStrings; values can be of any CFType (read the header)

Stream Properties

- Look in the appropriate header for supported properties
    - <CoreFoundation/CFStream.h>
    - <CFNetwork/CFSocketStream.h> for sockets
    - <CFNetwork/CFHTTPStream.h> for HTTP
- CFRead/WriteStreamCopyProperty() returns NULL if the property isn't recognized
- CFRead/WriteStreamSetProperty() returns FALSE
    - If the property isn't recognized
    - If configuration isn't allowed at that point
    - Set properties before opening the stream!

Streams and Threads

- The stream APIs are thread-safe, **but . . .**
- Individual streams are not
- If you want to use the same stream from multiple threads, you must lock

Streams and Multiple Run Loops

- You may schedule a stream on multiple run loops
- Events are dispatched in first-come, first-served fashion
- Events will not be duplicated across all run loops
- Locking is still your responsibility

Demo

Echo Server

For More Information

- Example code in **/Developer/Examples/Networking**
- Apple DTS-hosted list **macnetworkprog@lists.apple.com**
- Get the notes from Session 805: Introducing CFNetwork

Q&A

Status Summary: 8 Run

Location: Automatic

irPort

Configure: Using DHCP

IP Address: 17.255.96.164
(Provided by DHCP Server)

Tom Weyer
Network and Communications Evangelist
weyer@apple.com

http://developer.apple.com/wwdc2002/urls.html

WWDC2002

WWDC2002

WWDC2002